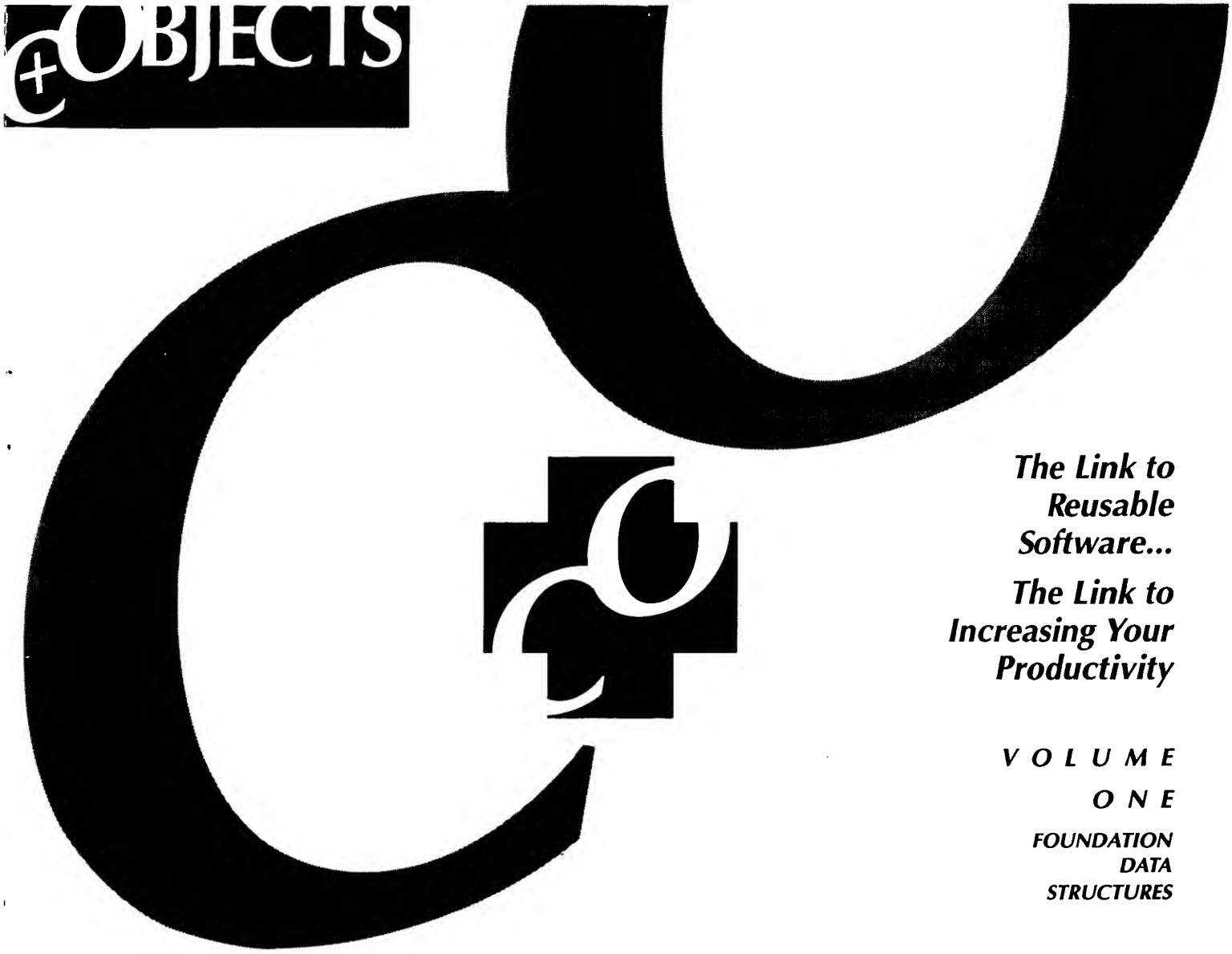


The Link to Reusable Software...

The Link to Increasing Your Productivity

VOLUME ONE

FOUNDATION DATA STRUCTURES

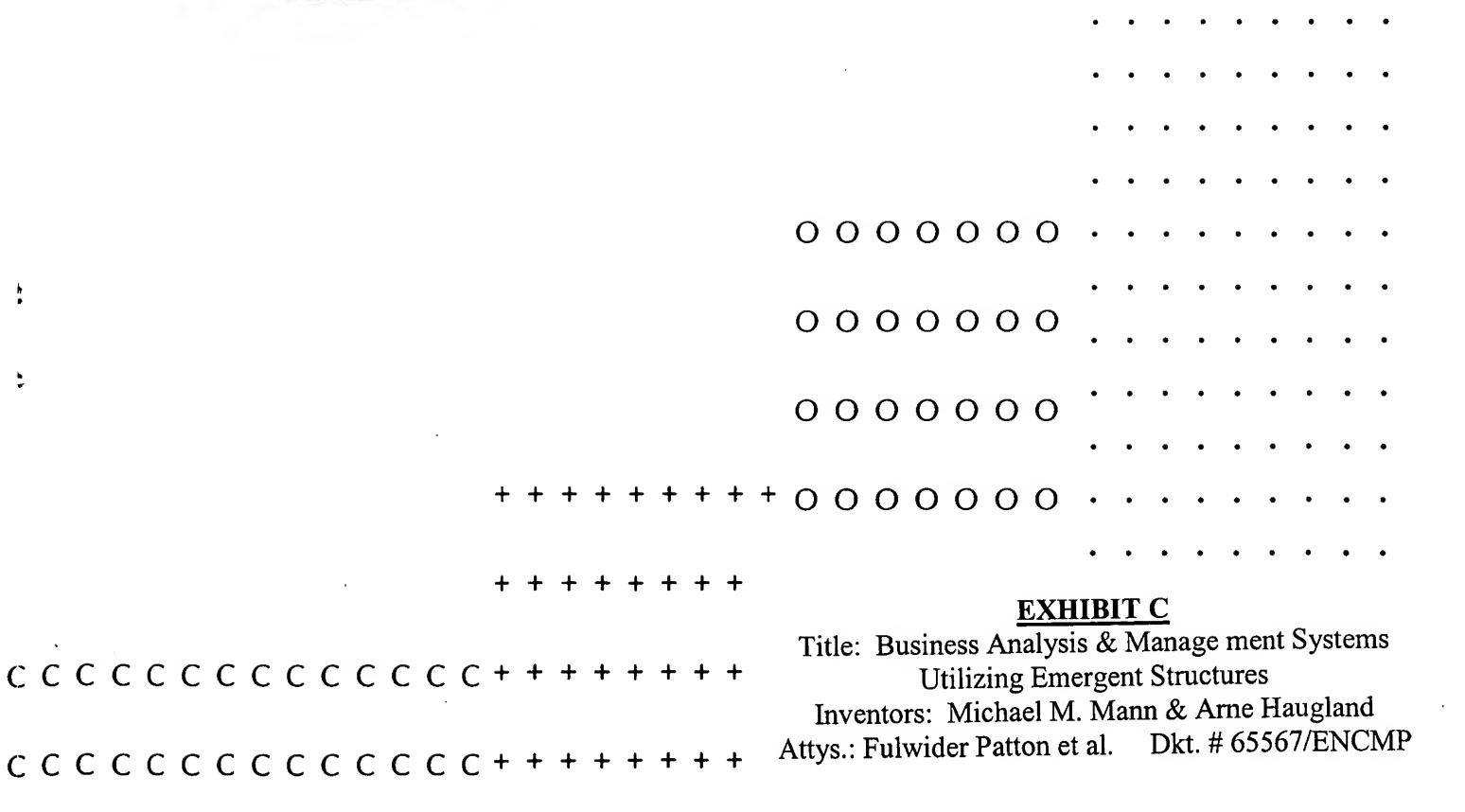


EXHIBIT C

Title: Business Analysis & Manage ment Systems Utilizing Emergent Structures

Inventors: Michael M. Mann & Arne Haugland

Attys.: Fulwider Patton et al. Dkt. # 65567/ENCMP

VOLUME 1 PRODUCT FEATURES

- Trees, Linked-Lists, Dynamic Arrays, Graphs, Strings, Dates, Objects, Classes
- Object-oriented design and implementation
- Written entirely in C
- Derive your own object types: Symbol Tables, Graphical Object Lists, Pars Trees, etc.
- Professional, fully tested code
- Advanced, multi-level exception-handler speeds coding and debugging
- Educational tool for data structures, object-oriented programming techniques and software engineering

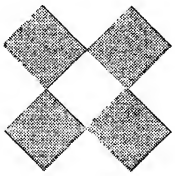
OOP FEATURES

- Each "class" is a C structure with related functions
- Objects are fully encapsulated
- Static and dynamic binding of "messages"
- New classes can "inherit" functionality and data from multiple object types
- Object-oriented control structures

WHAT'S INCLUDED

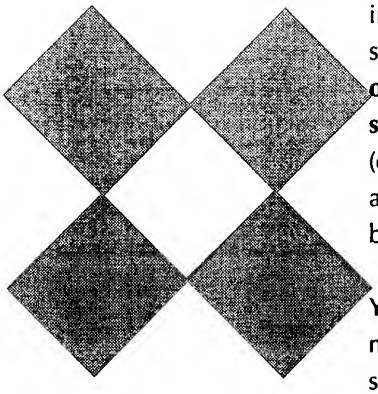
- 14 types of object, over 300 functions
- User's Guide explains object-oriented programming techniques, deriving your own object types, and includes tutorials
- Reference Guide with detailed information on each object type and function
- Demo and example programs
- Full source code available as option
- Debugging and production versions of libraries
- Support hot-line
- 30 day, money back guarantee

C+OBJECTS™ is a portable, object-oriented C function library used to reduce the investment required to build complex software.



What can C+OBJECTS do for me?

It can give you more creative time to design programs because you'll spend less time coding and debugging them. That's because the fundamental data structures used in many programs have already been built for you. Volume 1 includes data structures such as **doubly-linked lists, trees, dynamic arrays and graphs**. Volume 2 includes additional data structures such as **outlines, hash tables, stacks and queues** (details on Volume 2 appear in a separate brochure).



Your programs will be more reliable with the sophisticated, multi-level exception-handler and debug libraries.
You also get Julian (date) and String object types in our object-oriented format. The Julian routines have many calculations not available in other products.

Can C+OBJECTS data structures be customized?

That's the whole idea! Customizing and extending the functions of C+OBJECTS data structures is simple. Just "inherit" functionality from one or more C+OBJECTS data structures and add your own code and data on top.

For example, you could use the Tree data structure as the foundation for a parse tree. Or you might build a data structure for maintaining a graphical display list using the Doubly-Linked List object type. If you were building a data-flow diagram editor as part of a CASE package, you would find Graph, Vertex, and Edge well suited to the task. The uses for C+OBJECTS structures are virtually unlimited!
Customizing or extending C+OBJECTS object types does not involve modifying or recompiling the C+OBJECTS code or structures. C+OBJECTS would not be a useful tool otherwise.

What do you mean by an "object-oriented" function library?

Just as structured programming and structured design principles are not language dependent, neither are the principles of object-oriented programming. When we designed C+OBJECTS, we took the fundamental object-oriented programming techniques and applied them to C. Other object-oriented

tools for C have mimicked the Smalltalk implementation, complete with all of Smalltalk's faults and inefficiencies—we didn't, we married the best of both worlds.

And Performance?

C+OBJECTS is written entirely in C and does not use pre-processors or interpreters. Performance is what sets C+OBJECTS apart from the others.

C+OBJECTS provides macros for many functions. This gives you all the advantages of encapsulation without the performance penalty of calling a function to do a simple task.

Additionally, the messaging and inheritance features are implemented in a manner *tailor-made for C*. The result is cleaner and more efficient than Smalltalk's mechanisms.

Can it help me debug my programs faster?

Yes! C+OBJECTS advanced debugging features allow you to create *reliable* programs and do so easier and more quickly than you thought possible.

First, C+OBJECTS uses function prototypes to catch simple errors at compile time involving incorrect type, wrong ordering or wrong number of parameters.

Second, C+OBJECTS can detect when it is being passed a NULL or uninitialized pointer, pointers to the wrong type, or pointers to structures which have been "garbaged". It also checks for illegal values in other parameters types.

Third, C+OBJECTS includes an advanced exception handler package. With it , you can set up a single (or multiple level) exception handler which traps exceptions generated by C+OBJECTS functions.

If an exception is raised, you can determine the type and where it occurred. You can then recover from the exception or abort, depending on which is most appropriate. Exception handlers can allow your program to be well behaved, even in the presence of bugs.

This advanced error detection technique can be used in your own code as well. No longer do your programs need to check status codes after each function call. This results in less coding yet more reliable programs.

Once your program has been debugged, you can use C+OBJECTS Production Libraries with macro functions . This eliminates most or all of the debugging checkpoints.

What else can it do to increase my productivity?

C+OBJECTS goes beyond conventional function libraries by supplying a complete set

of *object-oriented control- structures.* These functions allow you to traverse data structures *without* having to use for, while, or do-while statements.

Control-structure functions simplify programs and eliminate a large number of potential errors — boundary conditions in loops for example.

Control-structure functions call a function of your choice for each item traversed. You can "inherit" these control-structure functions in your own data structures or create your own.

How portable Is C+OBJECTS?

C+OBJECTS was designed for portability to any operating system. Expect to see versions for Windows, OS/2, Presentation Manager, and Macintosh soon.

Is it suitable as an educational tool?

Yes. As an educational aid, it can teach you the principles of object-oriented programming. The User's Guide explains object-oriented programming and the differences between C+OBJECTS and Smalltalk. It could even be used as a primer for C programmers who wish to understand more about Smalltalk.

It can teach students the concepts of abstract data types and basic data structures. The linked list, tree, and graph types could form the foundation for a data structures class.

A software engineering course would benefit from a study of C+OBJECTS. It demonstrates good design principles, strict naming and portability conventions, and *defensive programming* techniques.

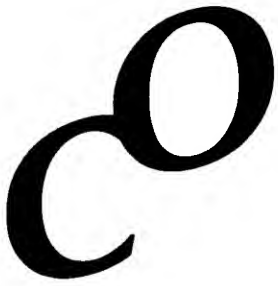
But don't let this fool you into thinking C+OBJECTS is *only* of educational value. C+OBJECTS is a serious development tool for professionals.

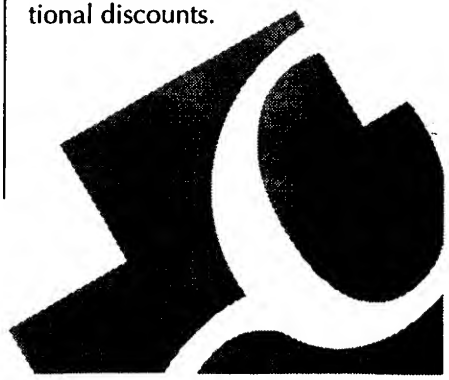
What about source code, royalties etc.?

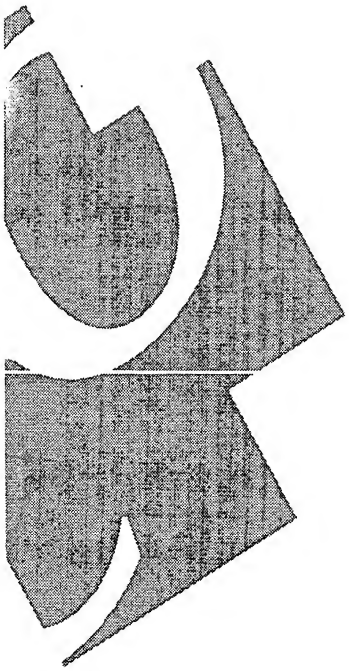
Full source code is available as an option. You will get more educational value out of C+OBJECTS with the source, but you don't need it to fully use or understand the product. Source will of course be necessary if you are porting C+OBJECTS to a new environment — call us first though, we may be able to help.

There are no royalties on programs developed using C+OBJECTS Volumes 1 or 2 and we do not require you to reproduce our copyright notice on your programs.

Call us for information on volume pricing, site licensing, and educational discounts.







Class

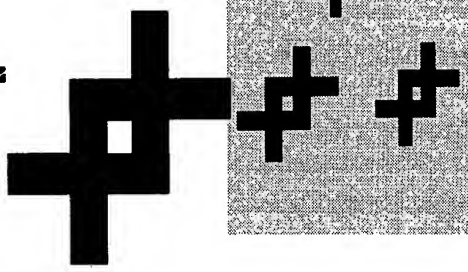
A Class (Cls) implements the object-oriented properties inheritance and messaging. It is used to subclass another object type. (See also Object page i i)

ClsDefaultInit	Initialize using defaults
ClsDestroyObj	Deallocate object
ClsGetClientOffset	Return client offset
ClsGetGpMsgFunc	Return (ptr.) message function ptr.
ClsGetMsgFunc	Return (int) message function ptr.
ClsInit	Initialize the class
ClsNewObj	Allocate object
ClsSetClientOffset	Set client offset
ClsSendObjMsg	Send message, return int
ClsSendObjGpMsg	Send message, return pointer

Doubly Linked List

A Doubly-Linked List (Dll) object is used to represent the head and tail of a linked list. A Dll contains objects of type List Element (or derivative objects). (See also List Element page 10)

DllAppend	Append element(s) to list
DllAppendLast	Append element(s) to end of list
DllAppendOne	Append one element to list
DllAsObj	Return list as object
DllClear	Clear list
DllClient	Return client of list
DllClientDo	Do function: all elements
DllClientDoBkwds	Do function: elements backwards
DllClientCount	Do function: count elements
DllClientFind	Do search function: all elements
DllClientFirst	Return client of first element
DllClientGetNth	Return Nth client
DllClientOrNull	Return client or null
DllClientLast	Return client of last element
DllCut	Cut element(s) from list
DllCutAll	Cut all elements from list
DllCutOne	Cut one element from list
DllDeInit	Deinitialize list
DllDestroy	Deinitialize list, free space
DllGetFirst	Return first element
DllGetLast	Return last element
DllGetNth	Return Nth element
DllIsEmpty	Is list empty?
DllInit	Initialize list
DllInsert	Insert element(s) in list



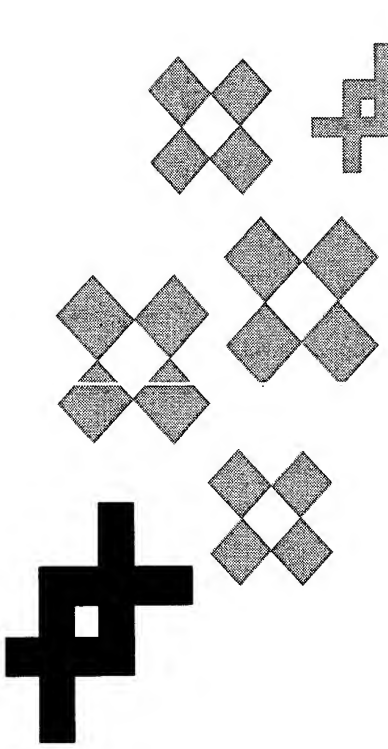
Doubly Linked List

DllInsertFirst	Insert elements first
DllInsertOne	Insert element in list
DllMakeFirst	Make element first
DllMakeLast	Make element last
DllNew	Initialize list object,allocate space

Dynamic Pointer Array

Dynamic Pointer Arrays (Dpa) are useful for storing arrays of pointers to objects of any type. A Dpa is dynamic because storage for the array is allocated and reallocated dynamically as the size of the array changes

DpaAppend	Append an element
DpaCut	Delete element(s)
DpaClear	Clear dynamic array
DpaCountTrueReturns	Do function: count True returns
DpaDeInit	Deinitialize dynamic array
DpaDestroy	Deinitialize object, free all memory
DpaDo	Do function: all elements
DpaDoRange	Do function: range of elements
DpaDoRangeCheckRet	Do function: range, check return
DpaDoRegion	Do function: region of elements
DpaDoSelfAndSuccessors	Do function: successors
DpaFindBkwd	Find index returning True
DpaFindFrwd	Find index returning True
DpaFindPtrBkwd	Find index with matching pointer
DpaFindPtrFrwd	Find index with matching pointer
DpaFindRangeFrwd	Find index returning True for range
DpaFindRangeBkwd	Find index returning True for range
DpaGetLast	Return last element in array
DpaGetNth	Return Nth array element
DpaGetSize	Return number of elements
DpaInit	Initialize dynamic array object
DpaLoad	Load array by looping function
DpaMakeElementsZero	Make range of elements null
DpaNew	Initialize object and allocate space
DpaPaste	Paste element(s) into array
DpaScrollDown	Scroll down N lines in array
DpaScrollUp	Scroll up N lines in array
DpaSetNth	Set Nth element of array
DpaSetSize	Set array size to N elements



Edge

An Edge (Edg) is used to represent a directed edge in a Graph (Grf). An edge can be connected and disconnected from two vertices (Vtx). An edge can belong to a single graph. (See also Vertex page 14 and Graph page 9)

EdgClientDo	Do function: edge
EdgConnectToVertices	Connect edge to vertices
EdgConnectToGrf	Connect edge to graph
EdgCompareInVtx	Compare vertex to incoming edge
EdgCompareOutVtx	Compare vertex with outgoing edge
EdgDeInit	Deinitialize the edge object
EdgDisconnectFromGrf	Disconnect edge from graph
EdgGetClient	Return client of edge
EdgGetGraphLel	Return as graph list element
EdgGetGrf	Return graph
EdgGetInLel	Return incoming edge list element
EdgGetInVtx	Return incoming vertex
EdgGetNextIn	Return next incoming edge
EdgGetNextOut	Return next outgoing edge
EdgGetOutLel	Return outgoing edge list element
EdgGetOutVtx	Return outgoing vertex
EdgGetVertices	Return vertices to edge
EdgInit	Initialize the edge object
EdgInGrf	Is edge in graph?
EdgNew	Initialize edge object and allocate
EdgSendDestroy	Send message for vertex destruction
EdgUpdateInVtx	Replace incoming vertex
EdgUpdateOutVtx	Replace outgoing vertex

Exception

An Exception (Exc) is a container for error/status information used when a program wants to raise an exception. An Exc contains the type of error, its location, and other pertinent information. Exceptions are invoked via a Thread (Thr).
(See also Threads page 12)

ExcClear	Clear exception
ExcDeInit	Deinitialize exception
ExcDestroy	Deinitialize exception, free space
ExcGetCode	Return error code
ExcGetFile	Return file where error detected
ExcGetLine	Return line where error detected
ExcGetOpSysErr	Return system error code
ExcGetType	Return type of error
ExcInit	Initialize exception
ExcIsFatal	Is exception non-recoverable?
ExcNew	Initialize exception, allocate space
ExcSet	Set exception fields

Graph

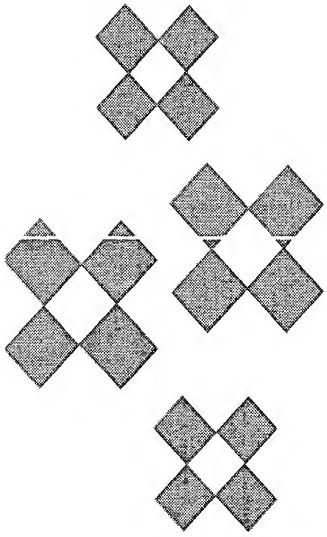
A Graph (Grf) object is used to represent a directed graph (or digraph) as understood by graph theory. A Graph is a collection of Vertices (Vtx) and (directed) Edges (Edg). A graph can be sorted topologically to determine if it is acyclic. (See also Vertex page 14 and Edge page 8)

GrfAddEdg	Add edge to graph
GrfAddVtx	Add vertex to graph
GrfClear	Clear the graph
GrfCountEdg	Count edges of graph
GrfCountVtx	Count vertices of graph
GrfDestroy	Deinitialize graph and free space
GrfDeInit	Deinitialize graph object
GrfDoEdgClient	Do function: edges
GrfDoVtxClient	Do function: vertices
GrfDoVtxInTopOrder	Do function: vertices in topological order
GrfInit	Initialize graph object
GrfNew	Initialize graph allocate space
GrfRemoveEdg	Remove edge from graph
GrfRemoveVtx	Remove vertex from graph
GrfTopologicalSort	Do topological sort of graph

Julian Date

A Julian Date (Jul) is used to represent a specific day in a specific year. The representation is purposely made explicit by its name. This representation of dates is most appropriate when date calculations are of more interest than formatting.

JulAddDays	Add/subtract days to date
JulAddDaysL	Add/subtract days to date (long)
JulAddMonths	Add/subtract months to date
JulAddQuarters	Add/subtract quarters to date
JulAddYears	Add/subtract years to date
JulToCalendar	Julian day to day, month, year
JulCopy	Copy julian day
JulDateStrToJulian	Date string to julian day
JulDaysInMonth	Days in month
JulDaysInQuarter	Days in quarter
JulDaysInYear	Days in year
JulDayOfYear	Day number in year
JulDayOfWeek	Day number in week
JulDiff	Days between two dates
JulDiffL	Days between two dates (long)
JulFromCalendar	Day, month, year to julian
JulGetMaxValue	dayReturn maximum julian value
JulGetSystemJulianDay	System date as julian day
JulInit	Initialize object
JulIsLeapYear	Is date in leap year?
JulIsMaxValue	Is date maximum julian value?



Julian Date

A Julian Date (Jul) is used to represent a specific day in a specific year. The representation is purposely made explicit by its name. This representation of dates is most appropriate when date calculations are of more interest than formatting. *(Continued from page 9)*

JulMax	The maximum of two julian dates
JulMin	The minimum of two julian dates
JulMonthDayDiff	Days between date and day/month
JulMonthString	Fill string with month and year
JulQuarterString	Fill string with quarter and year
JulSameDayMonth	Are dates same day and month?
JulSetMaxDate	Set date to maximum value
JulToDateStr	Fill date string with specified format
JulValidateDate	Validate date passed as string
JulWeekString	Fill string with week
JulYearString	Fill string with year

List Element

A List Element (Lel) object is used to maintain membership in a doubly-linked list (Dll). A Lel knows its previous and next list elements and the list it belongs to, if any. *(See also Doubly-Linked List page 6)*

LelAppend	Append elements(s) to list
LelAsObj	Return element as object
LelClientDll	Return client of list
LelClientNext	Return client of next element
LelClientPrev	Return client of previous element
LelClientCountSelfAndSuccessors	Return count of successors
LelClientDoSelfAndPredecessors	Do function: predecessors
LelClientDoSelfAndSuccessors	Do function: successors
LelClientDoPredecessors	Do function: predecessors
LelClientDoSuccessors	Do function: successors
LelClientDoRange	Do function: range
LelClientFindRange	Do search function: range
LelCount	Count elements
LelCut	Cut element(s) from list
LelDeInit	Deinitialize list element object
LelDoRange	Do function: for range
LelElementsAreInOrder	Are two elements in order?
LelGetClient	Return client
LelGetDll	Return list object is in
LelGetNthSuccessor	Return Nth successor element
LelGetNext	Return next element
LelGetPrev	Return previous element
LelInit	Initialize list element object
LelInList	Is element in list?
LelInsert	Insert element(s) to list
LelMakeList	Make elements into list

Object

An Object (Obj) implements the object-oriented properties of inheritance and messaging. It is of use for implementing reusable data types (as opposed to application-specific types). (See also Class page 6)

ObjDeInit	Deinitialize object
ObjDestroyClient	Deallocate object
ObjGetClientOrNull	Return client
ObjGetGpMsgFunc	Return (ptr.) message function
ObjGetMsgFunc	Return (int) message function
ObjGetClient	Return client of subclass
ObjInit	Initialize object
ObjSetClient	Set client
ObjSendClientGpMsg	Send client a (ptr.) message
ObjSendClientMsg	Send client a (int) message

String

The String (Str) class is used to represent null terminated character arrays.

StrAsMediumInt	String to 16 bit integer
StrExtract	Extract substring from string
StrFromMediumInt	Integer to string
StrHash	Return hash value of string
StrReplaceSubStr	Replace substring in string
StrSqueez	Removes any character from string
StrToLower	Change case of string to lower
StrToUpper	Change case of string to upper

Task

A Task (Tsk) object is used to represent a program. A Tsk owns all the threads in that task (one in MS-DOS). A task contains information used to invoke the program and other global information which belongs to a task. (See also Thread page 12 and Exception page 8)

TskDeInit	Deinitialize task
TskDestroy	Deinitialize task and free space
TskExit	Exit task with code
TskExitWithMsg	Exit task after displaying message
TskInit	Initialize task
TskNew	Initialize task and allocate space

Thread

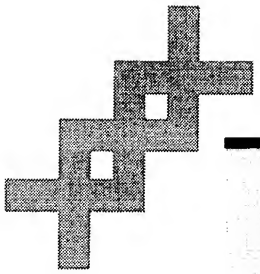
A Thread (Thr) is used to represent a single thread-of-control (similar to OS/2). However, MS-DOS implements only single threaded programs, therefore there is only one instance of a Thr. The only use threads have currently, is as a mechanism for pushing, popping, and invoking exception handlers (in the Ada style). Typically, a program might set up a single exception handler via Thr which traps any program logic errors (are triggered with "asserts"). (See also Task page 11 and Exception page 8)

ThrBadParameter	Signal bad function parameter
ThrClear	Clear thread
ThrDisablePushAndPop	Disable further signaling
ThrDiskFull	Signal disk full
ThrDeInit	Deinitialize thread object
ThrEndOfFile	Signal end of file
ThrEnablePushAndPop	Enable signaling
ThrFatalLogicError	Signal program logic error
ThrInit	Initialize thread object
ThrIsFatalError	Is exception non-recoverable?
ThrIsInitialized	Is thread initialized?
ThrOpSysError	Signal system error
ThrOutOfMemory	Signal out of memory
ThrPopCtx	Pop to previous exception handler
ThrPushErfAndReturn	Invoke current exception handler
ThrPushCtx	Push new exception handler
ThrReturnStatus	Signal status condition
ThrWarning	Signal warning

Tree

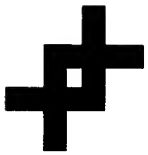
A tree is a recursive data structure that may contain zero or more children trees and zero or one parent trees.

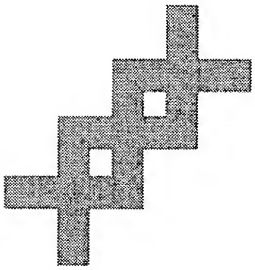
TreAppChild	Append child(ren)
TreAppSibling	Append sibling(s)
TreAsDll	Return tree as linked list
TreAsLel	Return tree as list element
TreAsObj	Return tree as object
TreClient	Return client of tree
TreClientNextSequential	Return next sequential client tree
TreClientDoAllSuccessors	Do function: all successors
TreClientDoBreadthFirst	Do function: breadth first
TreClientDoBranchDepthFirst	Do function: branch depth first
TreClientDoChildren	Do function: children, forwards
TreClientDoChildrenBkwds	Do function: children, backwards
TreClientDoDepthFirst	Do function: depth first
TreClientDoDepthFirstBkwds	Do function: depth first, backwards
TreClientDoDescBranchDepthFirst	Do function: descendent branches
TreClientDoDescBreadthFirst	Do function: descendent breadth
TreClientDoDescDepthFirst	Do function: descendent depth
TreClientDoDescDepthFirstBkwds	Do function: descendent depth
TreClientDoDescLeaves	Do function: descendent leaves
TreClientDoLeaves	Do function: leaves



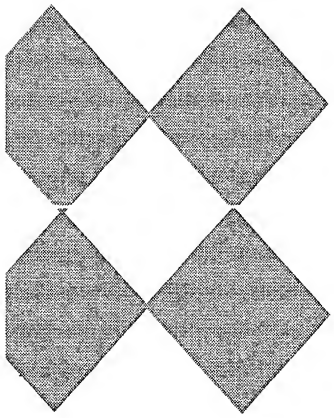
Tree

A tree is a recursive data structure that may contain zero or more children trees and zero or one parent trees.
(Continued from page 12)





TreClientDoParentsNearestFirst	Do function: nearest parents first
TreClientDoRange	Do function: range
TreClientDoSuccessors	Do function: successors
TreClientFindChild	Do search function: children
TreClientFirstChild	Return client of first child
TreClientLastChild	Return client of last child
TreClientLastLeaf	Return client of last leaf
TreClientNext	Return client of next sibling
TreClientNextUncle	Return client of next uncle
TreClientParent	Return client of parent
TreClientPrev	Return client of previous sibling
TreClientPrevSequential	Return previous client sequentially
TreCut	Cut node(s) from tree
TreCutChildren	Cut children from tree
TreDeInit	Deinitialize tree object
TreDoAllSuccessors	Do function: successors
TreDoBranchDepthFirst	Do function: branches depth first
TreDoBreadthFirst	Do function: breadth first
TreDoChildren	Do function: children
TreDoChildrenBkwds	Do function: children backwards
TreDoDepthFirst	Do function: depth first
TreDoDepthFirstBkwds	Do function: depth first backwards
TreDoDescBranchDepthFirst	Do function: descendent branches
TreDoDescBreadthFirst	Do function: descendent breadth
TreDoDescDepthFirst	Do function: descendent depth
TreDoDescDepthFirstBkwds	Do function: descendent depth
TreDoDescLeaves	Do function: descendent leaves
TreDoLeaves	Do function: leaves
TreDoRange	Do function: range
TreDoSuccessors	Do function: successors
TreFirstChild	Return first child
TreHasChildren	Does tree have any children?
TreHasSiblings	Does tree have any siblings?
TreInit	Initialize tree object
TreIsChild	Is tree a child?
TreIsDirectAncestor	Is related related to another?
TreIsRoot	Is tree the root?
TreInsChild	Insert child(ren)
TreInsSibling	Insert sibling(s)
TreLastChild	Return last child



Tree

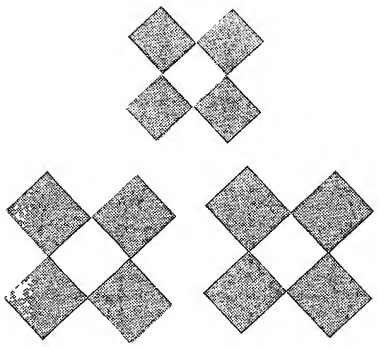
A tree is a recursive data structure that may contain zero or more children trees and zero or one parent trees. *(Continued from page 13)*

TreLastLeaf	Return last leaf
TreNew	Allocate and initialize tree object
TreNext	Return next sibling
TreNextSequential	Return next sequential tree
TreNextUncle	Return next uncle
TreParent	Return parent
TrePrev	Return previous sibling
TrePrevSequential	Return previous sequential tree
TreSendMsg	Send a int message to client
TreSendGpMsg	Send a ptr. message to client

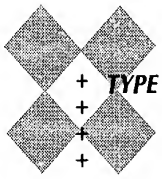
Vertex

A Vertex (Vtx) is used to represent a node in a directed graph (Grf). A vertex can belong to a single graph. It can access each of its incoming (arrow-end) edges (Edg) and each of its outgoing edges. It can also access all its predecessor vertices and successor vertices. *(See also Graph page 9 and Edge page 8)*

VtxAddInEdg	Add incoming edge
VtxAddOutEdg	Add outgoing edge
VtxClear	Clear vertex
VtxConnectToGrf	Connect vertex to graph
VtxCountIn	Count incoming edges
VtxCountOut	Count outgoing edges
VtxDisconnect	Disconnect vertex from graph
VtxDoEdge	Do function: all edges
VtxDoEdgeClients	Do function: clients of all edges
VtxDoInEdge	Do function: incoming edges
VtxDoInEdgeClient	Do function: incoming edges
VtxDoOutEdge	Do function: outgoing edges
VtxDoOutEdgeClient	Do function: outgoing edges
VtxDeInit	Deinitialize vertex object
VtxDestroy	Deinitialize vertex object and free
VtxDisconnectFromGrf	Disconnect vertex from graph
VtxFindOutEdg	Do search function: outgoing edges
VtxFindOutEdgClient	Do search function: outgoing edges
VtxGetClient	Return client of vertex
VtxGetFirstIn	Return first incoming edge
VtxGetFirstOut	Return first outgoing edge
VtxGetGraphLel	Return as list element in graph
VtxGetGrf	Return graph
VtxInGrf	Is vertex in graph?
VtxInit	Initialize vertex object
VtxNew	Initialize vertex, allocate space
VtxRemoveInEdg	Remove incoming edge
VtxRemoveOutEdg	Remove outgoing edge
VtxSendClientMsg	Send message to client



```
/* The following program fragment demonstrates inheritance from the Tree object type.  It is not complete
  but is representative of the use of C+OBJECTS*/

struct Node {                           /* Node is a specialized kind of Tree */
  char    *name;                        /* Name for each node */
  Tree    tre;                          /* Node inherits from Tree */
}; typedef  struct Node  Node;
Node    *pNodR = {0};          /* The root node */
Class   NodeCls = {0}, *NodTreCls = &NodeCls; /* To inherit from Tree, we need a "class" describing Node */

int   main() {
  NodInitializeModule();                /* Initialize classes */
  NodBuildTree();                       /* Create a sample set of tree nodes */

/*   "TreClientDo" functions will call a function, NodPrint in these examples,
     and pass the "client" of the tree, a Node pointer in this case, for each tree/node visited */
/*   Print the children nodes of root: a b c */
  TreClientDoChildren( &pNodR->tre, NodPrint ); printf( "\n" ); /* Object-oriented control-structure */
/*   Print the nodes in depth first order: root a.1 a.2 a b c */
  TreClientDoDepthFirst( &pNodR->tre, NodPrint ); printf( "\n" ); /* Object-oriented control-structure */
}

void  NodBuildTree( void ) {            /* Builds a sample tree of nodes */
  Node *pNod, *pNoda;
  pNodR = NodNew( "root" );             /* Create the root node */
  pNoda = NodNew( "a" );     NodAppChild( pNodR, pNod );
  pNod  = NodNew( "b" );     NodAppChild( pNodR, pNod );
  pNod  = NodNew( "c" );     NodAppChild( pNodR, pNod );
  pNod  = NodNew( "a.1" );   NodAppChild( pNoda, pNod ); /* Note: we are adding to pNoda */
  pNod  = NodNew( "a.2" );   NodAppChild( pNoda, pNod ); /* Ditto */
}

Node  *NodNew( char *name ) {           /* Allocate memory for a new node and initialize it */
  Node *pNod;
  pNod = (Node *) malloc( sizeof (Node) );
  TreInit( &pNod->tre, NodTreCls, (char *) pNod ); /* Initialize the tree.  TreInit needs a class and instance: the "client" */
  pNod->name = name;
}

void  NodAppChild( Node *pNodP, Node *pNodC ) {  /* Inherit the Tree function TreAppChild */
  TreAppChild( &pNodP->tre, &pNodC->tre, &pNodC->tre );  /* This adds pNodC as the last child of the parent pNodP */
}

void  NodPrint( Node *pNod ) {  /* Print a node name given a Node pointer */
  printf( "%s ", pNod->name );
}

void  NodInitializeModule( void ) { /* Initialize the class which describes Nodes */
  ClsDefaultInit( NodTreCls );      /* DefaultInit uses a default class description */
}
```



TYPE OF LIBRARY:	Object-Oriented Data Structures, Abstract Data Types, Exception Handler, Date and String
Number of Classes:	18
Number of Functions:	over 300
Compilers:	Microsoft C 5.0+ Quick C 2.0+ Turbo C 2.0
Operating Environments:	DOS Windows OS/2 Xenix Sun Unix
Memory Models:	All models
Version:	2.0

C+Objects is engineered and published by
Objective Systems
Architects and Engineers of Software
2443 Fillmore Street
San Francisco / California / 94115

PHONE 415/ 929-0964 FAX 415/ 929-8015

C C

C C C